NASA Contractor Report 191496

ICASE Report No. 93-39

ICASE

20 Years of Excellence

PARAMETRIC BINARY DISSECTION

Shahid H. Bokhari
Thomas W. Crockett
David M. Nicol

NASA Contract Nos. NAS1-19480, NAS1-18605
July 1993

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23681-0001

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

PARAMETRIC BINARY DISSECTION[1]

Shahid H. Bokhari[2]
Department of Electrical Engineering
University of Engineering & Technology
Lahore, Pakistan

Thomas W. Crockett
ICASE, NASA Langley Research Center
Hampton, Virginia

David M. Nicol[3]
Department of Computer Science
College of William & Mary
Williamsburg, Virginia

ABSTRACT

Binary dissection is widely used to partition non-uniform domains over parallel computers. This algorithm does not consider the perimeter, surface area, or aspect ratio of the regions being generated and can yield decompositions that have poor communication to computation ratio.

Parametric Binary Dissection (PBD) is a new algorithm in which each cut is chosen to minimize $load + \lambda \times (shape)$. In a 2 (or 3) dimensional problem, *load* is the amount of computation to be performed in a subregion and shape could refer to the perimeter (respectively surface) of that subregion. *Shape* is a measure of communication overhead and the parameter λ permits us to trade off load imbalance against communication overhead. When λ is zero, the algorithm reduces to plain binary dissection.

This algorithm can be used to partition graphs embedded in 2 or 3-d. Here load is the number of nodes in a subregion, shape the number of edges that leave that subregion, and λ the ratio of time to communicate over an edge to the time to compute at a node. We present an algorithm that finds the depth d parametric dissection of an embedded graph with n vertices and e edges in $O(\max[n \log n, de])$ time, which is an improvement over the $O(dn \log n)$ time of plain binary dissection. We also present parallel versions of this algorithm; the best of these requires $O((n/p) \log^3 p)$ time on a p processor hypercube, assuming graphs of bounded degree.

We describe how PBD is applied to 3-d unstructured meshes and yields partitions that are better than those obtained by plain dissection. We also discuss its application to the *color image quantization* problem, in which samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error.

[1]Research Supported by NASA Contract No. NAS1-19480, while the authors were in residence at the Institute for Computer Applications in Science & Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.

[2]Research supported by a grant from the Directorate of Research Extension and Advisory Services, University of Engineering & Technology, Lahore, Pakistan.

[3]Research supported in part by NSF grant CCR-9201195

1 Introduction

The partitioning of problems over the processors of a parallel computer system remains the subject of considerable research. This problem is particularly difficult when the domain or region being partitioned has nonuniform computational requirements. For example, in a climate model, some areas of the earth's surface may require greater computational effort than others. We would like to apportion parts of the problem domain over the processors of the system in such a way as to put equal computational load on all processors, so as to minimize the total computational time. Another example is the solution of aerodynamic problems using 'unstructured' meshes which are graphs embedded in 2 or 3-dimensional space[1]. Such meshes are increasingly being used to investigate the aerodynamic properties of aircraft.

Problems of this type require huge amounts of computational power and are at the limits of the memory capacities of the largest parallel processors. There is a pressing need for techniques to improve the running time of such problems, because they require scarce and expensive resources for their solution and also because the solution itself has great economic value. The solution to a weather calculation obviously decreases in value the longer it takes to compute. For the case of physical calculations based on unstructured meshes, a fast solution technique permits the designer to evaluate a larger number of alternatives within the course of a single session.

The binary dissection or orthogonal recursive partition algorithm was developed by Berger & Bokhari in 1985 [3, 4] as a means for partitioning non-uniform domains. It was inspired by Bentley's work on k-dimensional search trees [2]. This approach permits a very fast solution to the partitioning problem and has found many applications [1, 9, 17]. The key idea behind this algorithm is to make a series of bisections, along orthogonal directions, minimizing the load imbalance at each step. This algorithm does not take into consideration the perimeter, surface area, or aspect ratio of the subregions being generated and can yield decompositions that have poor communication to computation ratio.

In the present paper we present a new *parametric binary dissection (PBD)* algorithm in which each recursive cut is chosen to minimize $load + \lambda \times (shape)$.

[1] As opposed to structured meshes which are basically cartesian grids, possibly with nonuniform spacing.

When the domain is made up of a 2-dimensional region, *shape* could refer to the perimeter of a subregion. In the 3-d case it could refer to the surface area. When this approach is applied to the problem of partitioning embedded graphs, *load* refers to the number of vertices in a region and *shape* the number of edges leaving a region. In general, *shape* is a measure of the communication overhead and the parameter λ permits us to trade off load imbalance against communication overhead. We can sacrifice some amount of load balance for better shape balance in order to obtain faster overall computation time. When λ is zero, the new algorithm reduces to simple binary dissection.

This algorithm finds applications to partitioning problems in fields other than parallel processing. One example is *color image quantization*, in which samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error. In our formulation of this problem, one is given a 3-dimensional Boolean grid in which some points are occupied and others are vacant. The objective is to partition this grid into regions such that (1) the total number of regions is bounded by some given maximum, and (2) the maximum distance between any two points within a region is minimized.

Mesh partitioning is one of the problems to which we apply our algorithm. A number of other partitioning algorithms have been proposed for this problem, and it is worthwhile to compare and contrast our approach with existing work. The previous work [12, 15, 21] is built around the notion of graph separators. In such a formulation a mesh is viewed as an undirected graph. An edge-separator is a set of edges that disconnects the graph into two nearly equal sized pieces. The goal of separator based approaches is to find separators of small size, thereby reducing the communication overhead. There are two principal differences between parametric binary dissection, and separator-based algorithms. PBD constrains all cuts to be straight lines, a constraint not imposed on the other methods. As a consequence, for certain problems and ranges of parameter values, the partitions produced by PBD on this application are almost certainly inferior. This deficiency is balanced by the fact that

- PBD is more general in its application (e.g., we see no easy way to use graph separators for the color image quantization problem),
- linear cut constraints arise naturally in a number of applications, and

- PBD is undoubtably the simplest, and likely fastest method among the alternatives.

Thus the quality of partitions produced by PBD on the specific problem of mesh partitioning is not the sole measure of its value.

In Section 2 we review the original binary dissection algorithm. The basic ideas underlying parametric binary dissection are discussed in Section 3. In Section 4 we present a fast algorithm for parametric binary dissection. This algorithm can also be used for ordinary binary dissection and is faster than the previously known algorithm. A simple parallel algorithm for parametric binary dissection is presented in Section 5. A more elaborate, and faster, parallel algorithm appears in Section 6. Sections 7 and 8 describe applications of parametric binary dissection to unstructured meshes and to image quantization, respectively. We present our conclusions in Section 9.

2 Binary Dissection

The original binary dissection algorithm proposed by Berger & Bokhari [3, 4] can be applied to a variety of situations. In the present paper we are concerned with the partitioning of 2, 3 (or possibly higher) dimensional domains containing n points specified by their $x, y, z, \ldots$ coordinates. These points can be bisected along the x direction by first sorting by the x coordinate and then finding the mid-point. This process is accomplished in $O(n \log n)$ time for the sorting and $O(n)$ time for splitting the list of points. The bisection process is then repeated along the y direction for the two subdomains and so on.

If the *depth of partitioning* (the number of times the bisection is carried out) is given by d, then the entire process takes time

$$O(\sum_{i=0}^{d-1}(2^i(n/2^i \log n/2^i) + n)) = O(dn \log n). \quad (1)$$

Since the depth of partitioning $d \leq \log n$, this results in $O(n \log^2 n)$ in the case of problems where the partitioning is carried out to large depths. However, in many problems of interest the depth of partition d is small compared to $\log n$ and it is more meaningful to use expression (1).

The basic bisection step described above can also be carried out using a fast ($O(n)$) median finding algorithm[5, 6]. The sorting step is eliminated and we are left with $O(dn)$ time. However the constants involved in the linear time median finding algorithm are large and this method remains of theoretical interest only.

Although binary dissection has found many applications, for example [1, 9, 17], it partitions only on the basis of numbers of points. It is insensitive to the distribution of points in space. As a result, in its attempt to equalize the number of points at each bisection, it can yield partitions which have poor aspect ratio (the ratio of largest to smallest sides). This phenomenon may be undesirable in specific applications.

When binary dissection is applied to the partitioning of graphs embedded in 2 or 3 dimensional space, as is the case in many important aerodynamic problems, the edge information (which determines the amount of information that needs to be communicated between points) is ignored. Thus while binary dissection can be (and has been) applied to such problems, the partitions obtained can sometimes be poor as far as the compute/communicate ratio is concerned.

3 Parametric Dissection

Parametric binary dissection remedies one of the shortcomings of the basic algorithm by explicitly taking the shapes of regions into account. Thus, if the problem is to partition a three dimensional region that contains a number of points, we minimize at each bisection step $load + \lambda \times (shape)$ for the two subregions.

By *load* we mean the number of points in each region—this is the quantity that plain binary dissection minimizes. *Shape* can refer to a variety of properties of regions. For example, if the problem is to partition a 2-dimensional region into subregions such that the resulting subregions are as square as possible, we may wish to use the perimeters of the resulting rectangles as our shape property. At each bisection step we would minimize the the number of points in each rectangle plus λ times their perimeters. The parameter λ permits us to trade off load imbalance against shape imbalance—by sacrificing some amount of load balance, we can improve the shape imbalance.

The preceding example can be extended in an obvious fashion to 3 or

higher dimensions. Various shape properties can be used. For example, in Section 8, we partition three dimensional space and the shape property is the length of the diagonal. In the following discussion we shall assume that the shape property can be computed easily, so that the analysis of time complexity of the algorithm is not affected by it. Nothing keeps us from using a complicated shape property, as long as we are willing to pay for the time required to compute it while carrying out binary dissection. For the problem of Section 8 we could choose to use the distance between the two most distant points in any region as our shape property, which is relatively more expensive to compute.

The discussion so far has been in terms of point problems, where we are given a collection of points in 2, 3 or higher dimensional space. A more complicated situation arises when we are given a graph embedded in 2, 3 or possibly higher dimensions. Here each point or node has associated with it a set of coordinates as well as an adjacency list. The objective here is more straightforward: each bisection is carried out to minimize $nodes + \lambda \times (edges\ cut)$.

Graph partitioning problems arise in many environments, most notably in the analysis of unstructured meshes (Section 7). When such meshes are partitioned and mapped onto parallel computers, the running time is modeled by

$$\max_{all\ regions} [nodes\ in\ region + \lambda \times (edges\ leaving\ region)]. \tag{2}$$

Here λ corresponds to the well-known *communicate to compute ratio* for the given parallel computer system, that is, the ratio of time required to fetch a datum from a remote processor to the time to compute on a datum on the local processor. The time given by (2) is normalized to the time required to compute on one point, assuming a uniform computation cost for each point. A more refined expression for the parallel computation time for partitioned graphs is

$$\max_{all\ regions} [nodes\ in\ region + \kappa(edges\ in\ region) + \lambda \times (edges\ leaving\ region)]. \tag{3}$$

In this case κ is the time to fetch information from a neighboring point in the grid if that point lies on the same processor and λ is is the time to fetch this information if the point lies on a remote processor. Both quantities are normalized in terms of time required to compute on a point.

For the case of point problems complexity of parametric binary dissection is unchanged at $O(dn \log n)$, where d is the depth of partitioning[2]. For graph problems, the complexity is $O(\max[dn \log n, de])$, since we have to look at all edges before splitting, at every depth of the partition.

4 Fast Parametric Dissection

A major factor contributing to the time complexity of the binary dissection algorithms presented in Sections 2 and 3 is repeated sorting at each level of partitioning. We now show how parametric binary dissection can be accomplished by sorting only once per dimension. The fast algorithm we present also improves the time required for plain binary dissection.

Our fast algorithm obtains its efficiency by sorting only once per dimension. A separate index list is created for each dimension. When a region is partitioned, all indices are split, so that the sublists corresponding to each subregion remain sorted. For purposes of exposition, we assume a 3-d graph partitioning problem and partition on the basis of expression (2) of Section 3. A simple modification to the procedure given below permits us to partition on the basis of expression (3) of Section 3. This modification does not affect the complexity of the solution.

Let us suppose that the the index lists for the x, y and z dimensions are stored in arrays xlist[],ylist[] and zlist[]. The subregion to be partitioned is stored in array positions L...U. This means that the x dimension index list extends from xlist[L] to xlist[U] and so on. The current depth of partitioning is depth. The coordinates of point i are stored in x[i], y[i], z[i]. The procedure for computing the parametric cut is as follows.

procedure PARAMETRIC_CUT(depth,L,U,x,y,z,xlist,ylist,zlist);

1. Sweep forward from i=L to U counting the edges that would leave the left hand region, if the left hand region was L to i (inclusive). Store the result in leftvec[i].

2. Sweep backwards from i=U down to L counting the edges that would leave the right hand region, if the right hand region was i to U (inclusive). Store the result in rightvec[i].

[2]Assuming that the shape property for a region takes negligible time to compute.

3. Sweep forward again from i=L to U to find the optimal split point:

   - the left hand region comprises L to i,
   - the right hand region comprises i+1 to U
   - the optimal split point SPLITPLACE is the value of i for which the objective MAX((i-L+1) + λ×(leftvec[i]), (U-i) + λ×(rightvec[i+1])) is minimum. The value x[SPLITPLACE] is SPLITVALUE.

4. The xlist has now been split into two parts, L to SPLITPLACE and SPLITPLACE+1 to U. The x coordinates of these points are already sorted since the original index list was sorted and has not been disturbed.

5. Split the ylist: sweep forward from i=L to U moving successive values of ylist[i] for which x[ylist[i]] $\leq$SPLITVALUE to the first part of the list (Figure 2 illustrates this for a 2-d problem). The remaining values are moved to the second part of the list.

6. Similarly split the zlist.

7. At this point all three indices xlist, ylist and zlist have been split so that elements [L..SPLITPLACE] of these lists contain the points in one of the subregions and [SPLITPLACE+1..U] those in the other. When accessed through these lists the x, y and z coordinates of the points are in sorted order.

8. Recursively cut for next depth but along next dimension
   if(depth>1) then
   PARAMETRIC_CUT(depth-1,L,SPLITPLACE,y,z,x,ylist,zlist,xlist)
   PARAMETRIC_CUT(depth-1,SPLITPLACE+1,U,y,z,x,ylist,zlist,xlist)
   endif;

end parametric_cut;

Figure 1 clarifies how leftvec and rightvec are computed. The vertical dashed lines in this figure show one possible SPLITPLACE. The value of leftvec for this SPLITPLACE is 5. This is because if the right hand region was chosen to be up to and including the node through which this dashed line passes, the number of edges leaving the left hand region would be 5. Similarly, if the

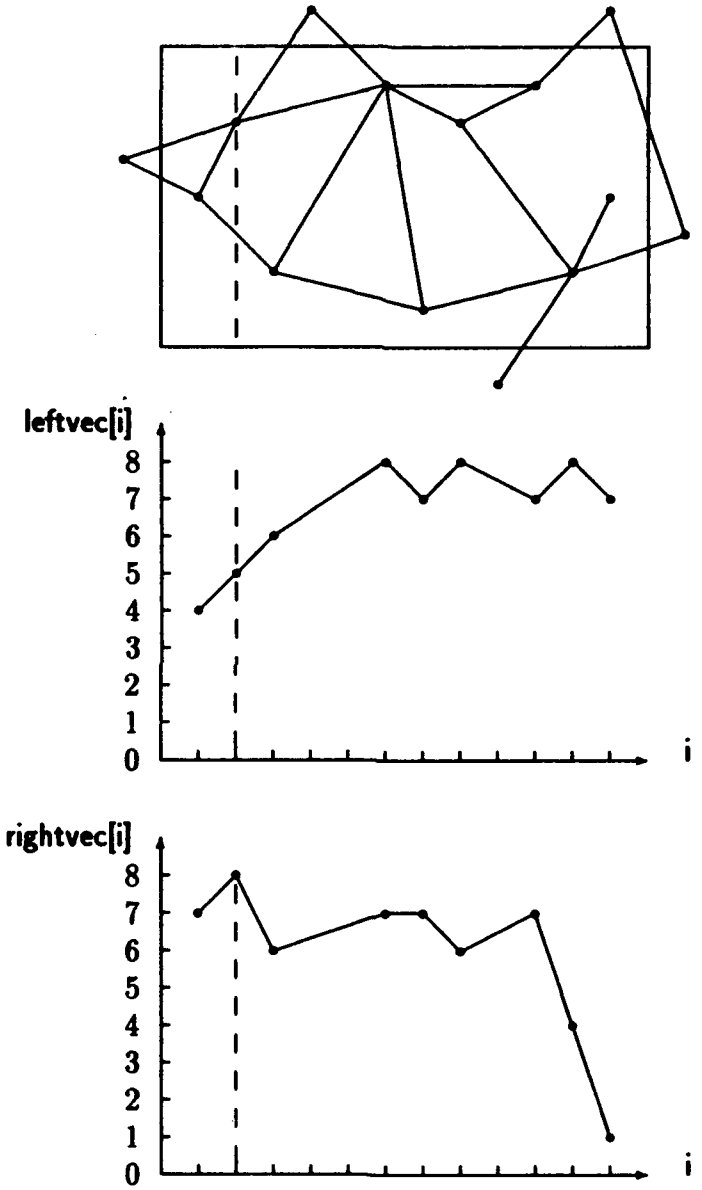

Figure 1: Computation of leftvec and rightvec in steps 1 and 2 of procedure parametric_cut. The domain to be partitioned (along the x-direction) is given by the top rectangle. The vertical dashed line is discussed in the text.

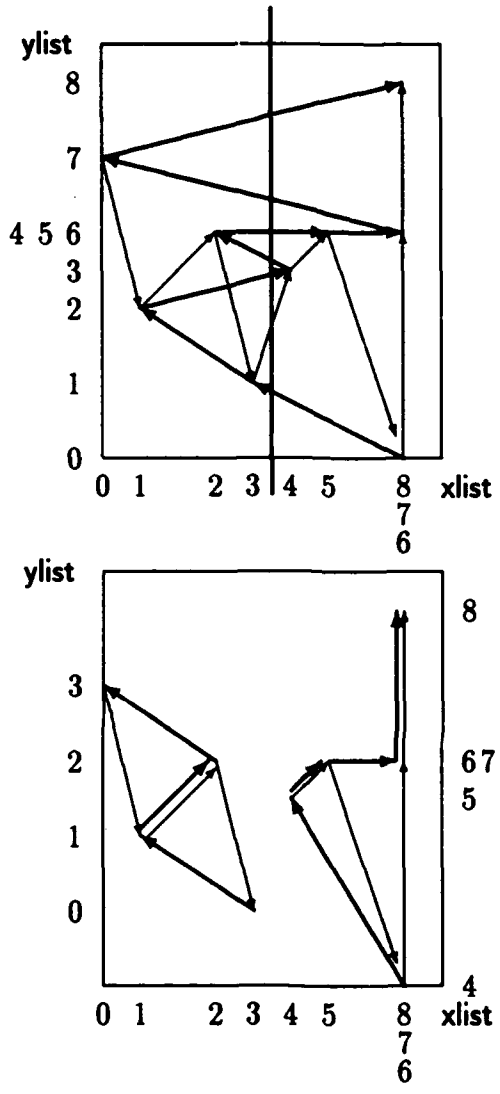

Figure 2: Splitting index lists: suppose we choose to split points in the domain as indicated by the vertical cut. xlist (thin arrows) is split in constant time. ylist (thick arrows) is split in time proportional to the number of points, since each point in this index list may have to be moved.

right hand region was chosen to start from this point onwards, the number of edges leaving the right hand region would be 8. Note that the edge lying wholly between points outside the region has no impact on the computation. Figure 2 shows how the index lists are split.

Assuming a fixed number of dimensions, the sorts take $O(n \log n)$ time. For point problems, each partition or split takes $O(n)$ time. We therefore get $O(n \log n) + O(dn) = O(n \log n)$ for point problems.

For graph problems the sorting time is unchanged. The time to split is now $O(e)$ per level, as we have to look at every edge at every level. The time is thus $O(\max[n \log n, de])$ for graph problems. However in this case it is important to remember that the graphs corresponding to unstructured grids from 2-d aerodynamic problems are planar and thus have $e = O(n)$. Typical 3-d aerodynamic grids have bounded degree [7] and again have $e = O(n)$. Thus we again obtain $O(n \log n)$.

5 A Simple Parallel Algorithm

We now discuss a parallel version of the parametric dissection algorithm. This is a simple algorithm that does not utilize the available processors well: its runtime is $O(n)$ *independent* of the number of processors, assuming that the data points are supplied in sorted form. However its extreme simplicity is likely to make its implementation easy and its measured run times may well be competitive with the more complex algorithm presented in Section 6. We start by considering point problems and discuss graph problems (which are only slightly more complicated to implement) at the end of this Section.

We make the reasonable assumption that the partitioning is to be carried out on the same parallel machine on which the problem is to be solved. Thus 2- and 3-d problems are computed on 2- and 3-d meshes, respectively. Alternatively, since a large enough hypercube can have any lower dimensional mesh embedded in it, we may choose to run our problems on hypercubes.

Discussion of a parallel implementation is complicated by the issue of *mapping*. Whereas in the serial algorithm we were only concerned with the partitioning, in the parallel algorithm we would like to partition our domain and at the same time deliver the resulting subdomains to the correct processors. This can result in substantial savings in time, as discussed below.

The question that now arises is how we are to map the 2^d subdomains

that arise after a depth d partitioning onto a $p = 2^d$ processor system. The mapping that we choose is the *natural mapping* described by Berger & Bokhari[3, 4]. When the first bisection is made, dividing the domain into, say, a left half and a right half, then the left subdomain is associated with the left half of the mesh and the right subdomain with the right half. This process is repeated until the subdomains at the dth level are reached—these are associated with individual processors.

5.1 Basic bisection step

We shall now suppose that we have a $p = 2^d$ processor *chain-connected* parallel machine. We shall describe how the basic bisection step is carried out on this chain and then later show how this chain is mapped onto the target parallel machine[3].

For purposes of illustration, we shall assume that we have a 2-d point problem with n points and that the point data (comprising $<x, y>$ coordinates) has been duplicated and two sorted lists prepared, one for each coordinate. These lists are loaded into our chain in a linear order, with $2n/p$ points per processor.

Sweep-x Sweep through each point of the x-list *sequentially* from left to right, in order to identify the optimal split point, as in Section 4. The x-coordinate of the split point is called SPLITVALUE.

Migrate-x Move all points of the x-list with x-coordinate $\leq$ SPLITVALUE to the left half of the processor chain and the remaining points move to the right half.

Mark-y Now sweep through the y-list, marking with the label LEFT, those points whose x-coordinates are $\leq$ SPLITVALUE and all others with RIGHT.

Migrate-y Move all points of the y-list marked LEFT to the left half of the processor chain and those marked RIGHT to the right half.

[3]Which could be a $2^{d/2} \times 2^{d/2}$ 2-d mesh, a $2^{d/3} \times 2^{d/3} \times 2^{d/3}$ 3-d mesh, or a dimension d hypercube.

Each of the above four steps takes time proportional to n. The basic bisection step can now be repeated on the two halves of the chain, with the roles of x and y interchanged. If at each bisection step the number of points is exactly halved, the time required is proportional to no more than

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < 2n.$$

Parametric binary dissection does not guarantee that each dissection step will exactly halve the number of points. We shall assume that the maximum number of points at every step of the partitioning is a constant times the ideal balance at that step. Thus the $O(n)$ result obtained above holds for plain as well as parametric binary dissection.

5.2 Bisectionable Chain Embedding

The bisection procedure described above only serves to partition the domain over a chain of processors. When carrying out computations on 2-d or 3-d domains we would naturally prefer to use 2- or 3-d meshes for our computation. We now describe embeddings of chains in 2- or 3-d meshes which have the interesting property that when the basic bisection step of Section 5.1 is successively applied to such chains, then the points migrate to the processors on which they should be mapped according to the *natural* mapping. No explicit routing of data blocks is required. This property eliminates an expensive routing step.

Figure 3 shows how a Bisectionable Chain Embedding (BCE) is constructed by combining two smaller BCEs. To formalize the rules for generating BCEs, note first that their sizes can be either $2^i \times 2^i$ or $2^i \times 2^{i+1}$, for some integer $i \geq 1$. The line segments making up BCES are parallel to either the x or the y axis. For a 'non-square' BCE (that is, one of size $2^i \times 2^{i+1}$), the longer side is parallel to the y axis. The rules for generating BCEs are as follows.

1. A BCE of size 2×2 is a square with corners at $(1,1)$, $(-1,1)$, $(1,-1)$ and $(-1,-1)$.

2. To construct a BCE of size $2^i \times 2^{i+1}$, translate a BCE of size $2^i \times 2^i$ so that its lower right corner lies at $(1,1)$. Reflect about the x axis. Delete

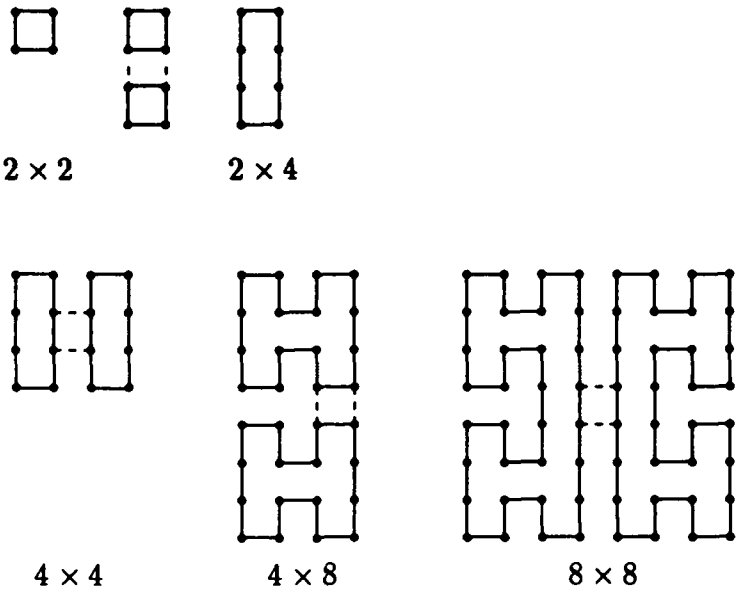

Figure 3: A Bisectionable Chain Embedding (BCE) of size 2×4 is constructed by juxtaposing two 2×2 BCEs and putting a 'bridge' between them. The figure shows how 4×4, 4×8 and 8×8 BCEs are constructed using this procedure.

---

the segments $(1,1) \leftrightarrow (-1,1)$ and $(1,-1) \leftrightarrow (-1,-1)$. Add segments $(1,1) \leftrightarrow (1,-1)$ and $(-1,1) \leftrightarrow (-1,-1)$.

3. To construct a BCE of size $2^{i+1} \times 2^{i+1}$, translate a BCE of size $2^i \times 2^{i+1}$ so that it is symmetric about the x axis and its rightmost edge lies on the line $x = -1$. Reflect about the y axis. Delete the segments $(1,1) \leftrightarrow (1,-1)$ and $(-1,1) \leftrightarrow (-1,-1)$. Add segments $(1,1) \leftrightarrow (-1,1)$ and $(1,-1) \leftrightarrow (-1,-1)$.

Figure 4 shows a bisectionable chain embedding of size 16×16. Our sorted x and y-lists are mapped onto this chain starting at $\bullet$ and ending at $\blacksquare$. If the basic bisection step is applied to these lists then we will obtain two

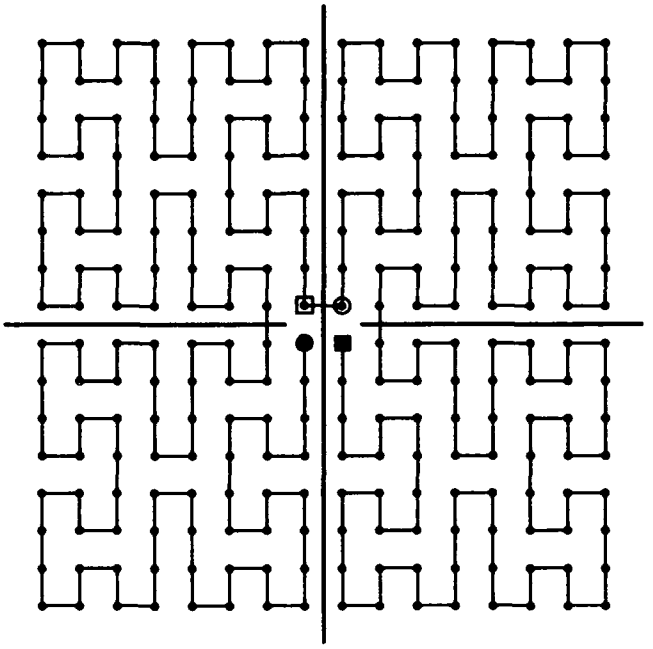

Figure 4: A bisectionable chain embedding (BCE) of size 16×16. The sorted x and y lists are mapped onto this chain starting at • and ending at ■. If the basic bisection step (vertical cut) is applied to these lists then we will obtain two sets of sublists, one set starting at • and ending at □; the other starting at ○ and ending at ■. This procedure can now be repeated with two horizontal cuts.

---

sets of sublists, one set starting at • and ending at □; the other starting at ○ and ending at ■.

The key property of BCEs is that at this stage the left half of the mesh chain will contain only the points of the original lists that should be mapped onto the left half of the mesh and similarly for the right half of the chain. Thus when the bisection procedure is carried out recursively on a BCE, the data points move to their respective parts of the mesh, so that at the end of the procedure each processor contains its naturally mapped points.

The concept of Bisectionable Chain Embeddings is easily extended to higher dimensions. Figure 5 shows a BCE for a $4 \times 4 \times 4$ 3-d mesh.

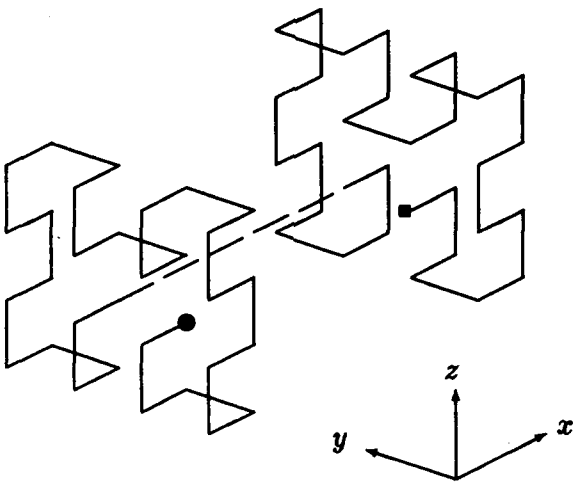

Figure 5: A 3-d BCE of size $4 \times 4 \times 4$. x, y and z lists are mapped onto this chain starting at • and ending at ■. The first bisection (with a plane perpendicular to the x axis) will split the BCE at the dashed segment. This procedure is then repeated recursively for the y and z directions. For clarity the spacing along the x axis has been distorted.

5.3 Graph Problems

We present our analysis for the case of degree constrained graphs embedded in 3-space, and assume that 3 copies of the graph are available to us, sorted by each of the dimensions[4]. Each item in the x-list, for example, contains the $<x, y, z>$ coordinates of the point and the coordinates of *all* points adjacent to this point. These lists are mapped onto a chain of processors as before

[4]Applications to higher or lower dimensions are immediate, although it is to be kept in mind that the space required by this algorithm (on each processor) is proportional to the number of dimensions of the problem. It should be recalled that the ultimate objective of the partitioning is to permit a complex aerodynamic computation to take place. The partitioning is carried out *before* the computation. The actual computation requires a large number of variables for each point to store, for example, the velocity vectors, pressure, density etc. Typically from 50 to 100 locations are required for each point [7]. This space can thus freely be used for the binary dissection.

and the chain of processors embedded in a 3-d mesh.

The basic bisection step for graph problems requires visiting each processor sequentially, and within each processor, traversing the x-list. As each point is visited, we count the number of edges that would be cut if this point were the extreme point in the bisection. This process is repeated in the reverse direction and then the point where the minimum of $nodes + \lambda \times (edges\ cut)$ occurs found along the lines of the serial procedure of Section 4. This is followed by list migration. This step is then repeated successively in the y and z directions.

Of course, at each point we must visit the nodes adjacent to that point, and list migration involves moving not only each point, but also its adjacent points. Our time complexity is unchanged at $O(n)$ because we have assumed a constant degree constraint.

6 Fast Parallel Algorithm

The ideal algorithm for parametric dissection of an n node problem on a p processor system would have complexity $O(n/p \log n/p)$, which is the same as if each processor were solving the subproblem resident on it in isolation. This lower bound is difficult to achieve because of the overhead of interprocessor communication (which depends heavily on the interconnection structure of the parallel processor). The fast algorithm that we present in this Section comes close to this bound, at least on hypercubes. We present our algorithm for a graph problem; application to the simpler point problem is straightforward.

6.1 Notation

We shall assume that the problem graph is made up of n nodes, with a fixed degree constraint. The problem graph is supplied to us in sorted form, one copy per dimension. The graph is initially partitioned into p blocks in a chain-like fashion; the chain is in turn embedded on our parallel processor according to the Binary Chain Embedding discussed earlier. The parallel processor may be interconnected as a 2- or 3-d mesh or as a hypercube[5]. To

[5] In the case of hypercubes, the 2- or 3-d BCE is embedded in a mesh which is, in turn, embedded in the hypercube using the Gray code embedding technique[16].

permit a unified analysis of our parallel algorithm, the times taken by certain required communication operations on the parallel processor are given by the symbols enumerated in Table 1. At the end of this Section we compare the performance of this algorithm on the three types of processor interconnects by substituting actual known expressions for these symbols.

Table 1: Times required on a k-processor system.

Symbol	Operation
$\mathcal{S}(k)$	sum-prefix
$\mathcal{C}(k)$	condense subchains
$\mathcal{M}(k)$	find minimum
$\mathcal{P}(k)$	arbitrary permutation

6.2 The Algorithm

For our exposition, we assume that an n node graph embedded in 2-d is given to us. Two copies of this graph, sorted by the x and y directions, are mapped onto a chain of p processors. The fast parallel parametric bisection algorithm has the following four steps.

1. Find leftvec and rightvec
2. Find optimal split point
3. Migrate the graph
4. Repeat recursively along next direction

6.2.1 Finding leftvec and rightvec in parallel

In order to find the optimal split point, we must compute leftvec and rightvec as was done in Section 4. Parallel computation of these vectors is complicated by the fact that the n node graph is distributed over p processors. The ideal lower bound of $O(n/p)$ is difficult to achieve because of the overhead of interprocessor communications.

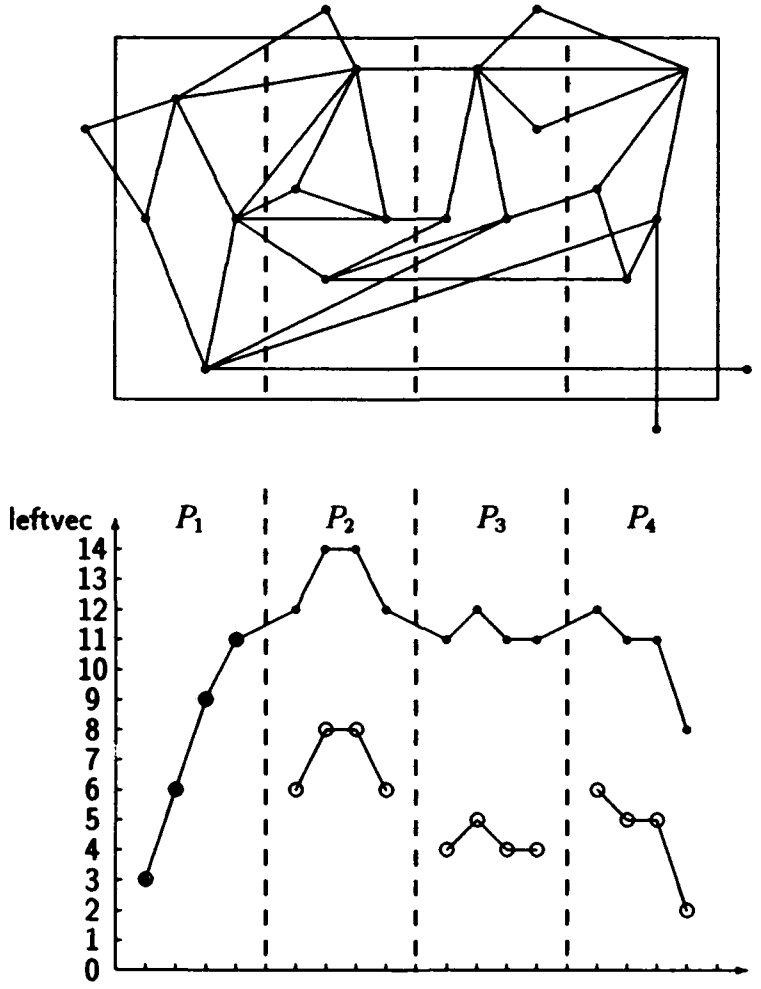

Figure 6: Parallel computation of **leftvec**. The domain to be bisected is given at the top and is partitioned across 4 processors. The upper plot (points marked •) shows the desired **leftvec**. The lower plots (points marked o) show local estimates of **leftvec** by each processor. Estimates on P_2, P_3, & P_4 are in error by 6, 7 & 6 units, respectively, because these processors do not know about edges that (1) straddle them or (2) leave the region from processors to their left.

Figure 6 shows a graph embedded in 2-d that has been partitioned across four processors, with n/p points per processor. The plot with points labeled • shows the desired **leftvec**. Each individual processor cannot compute this vector because it lacks crucial information about edges that are incident on nodes assigned to other processors that influence the vector positions that lie on itself.

Each processor can compute a local *estimate* of its portion of **leftvec** in $O(n/p)$ time. This is done by sweeping[6] through the points in x order, adding up in **local_leftvec[i]** the edges that would leave the region if the bisection point was chosen to be just beyond node i. This process takes $O(n/p)$ time, since each processor has only to compute its n/p elements of **local_leftvec**.

In the example of Figure 6, the **local_leftvecs** on processors P_2, P_3 & P_4 are in error. On P_2 for example, the estimate is consistently 6 units below the desired value. This is because P_2 is not aware of the 3 edges that straddle it, and the three edges that leave the region from nodes within processor P_1. It is thus clear that a communication step is required to inform all processors of (1) all straddling edges and (2) all edges that leave the region from other processors.

For every processor P_k, $k > 1$, define L_k to be the number of edges that are cut by the separation of the chain between processors P_k and P_{k-1}. It is important to remember that the processor ordering is with respect to the embedded chain. Given L_k, P_k can compute **leftvec[i]** = **local_leftvec[i]** + L_k for all the points i resident on P_k. The problem then is to determine the set of values L_k, in parallel.

Each processor P_k can count the total number of edges that have an endpoint in P_k and an endpoint in any processor to the left of P_k (i.e., in some P_j, $j < k$). Denote this total by I_k. Similarly, each P_k can count the total number of edges that have an endpoint in P_k and an endpoint in any processor to the right of P_k; call this O_k. Now observe that the number of edges that span P_k is $L_k - I_k$: the number that enter P_k from the left, less the edges that terminate in P_k. The number of edges that are cut by the P_k to P_{k+1} split (i.e., L_{k+1}) is equal to the number of edges that span P_k, plus the number of edges that originate in P_k: $(L_k - I_k) + O_k$. This gives us the

[6]This sweep is slightly more complicated than the sweep for the serial algorithm, since edges can be encountered at node j that increase the values of **local_leftvec[i]** for all $i < j$. However, it can still be accomplished in $O(n/p)$ time.

recursion

$$\begin{aligned} L_{k+1} &= L_k - I_k + O_k \\ &= L_k + D_k \qquad \text{where } D_k = -I_k + O_k \\ &= \sum_{i=1}^{k} D_i. \end{aligned}$$

From this we see that the problem of computing the values of L_k is solved simply by a parallel sum-prefix on the values D_k, which we assume to take time $\mathcal{S}(k)$. The total time required in finding **leftvec** and **rightvec** is thus $O(n/p) + \mathcal{S}(k)$.

6.2.2 The optimal split point

Once the global information has been compensated for, each processor has a portion (of size n/p) of **leftvec** resident on it. It now remains to find the optimal split point. Each processor can find its local minimum in time $O(n/p)$ and all processors can decide on the global minimum in time $\mathcal{M}(p)$.

6.2.3 Migrating the graph

The next step is the migration of the x and y copies of the graphs to the appropriate halves of the parallel processor. As far as the x copy is concerned, the split point is already known. This is illustrated in Figure 7. In this Figure a graph of 16 nodes is distributed uniformly over 4 processors. The bisection point in this example happens to assign 10 nodes to the left half of the domain and 6 nodes to the right half. After migration, therefore, there must be 5 nodes each on processors P_1 & P_2 and 3 nodes each on P_3 & P_4. Solid vertical lines indicate the original partition while dashed lines indicate the partition after migration. The migration of the x graph requires the movement of no more than n/p points from a processor to its neighboring processors.[7] The migration of the x graph thus takes time $O(n/p)$.

The migration of the y copy of the graph is far more complex and is illustrated in Figure 8. This Figure shows the y copy of the graph of Figure 7. This y copy is also partitioned uniformly over four processors, as indicated by the solid vertical lines. Note that the x and y axes are interchanged in

[7] This is because the vertical strips of the domain are mapped onto a chain of processors.

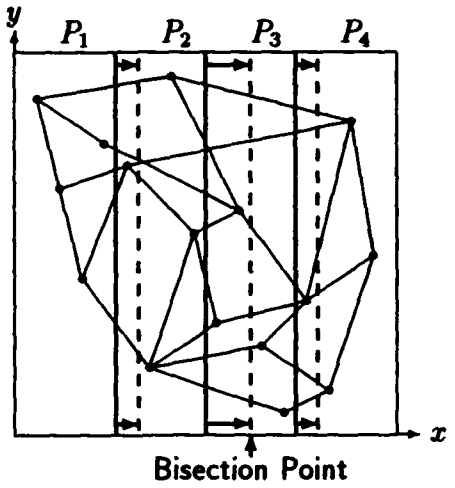

Figure 7: A 16 node graph, sorted by the x direction, uniformly partitioned over four processors (thick solid lines). The bisection assigns 10 (6) nodes to the left (right) half of the domain. Nodes must be migrated so as to place 5 nodes each on $P_1 - P_2$ and 3 nodes each on $P_3 - P_4$ (dashed lines).

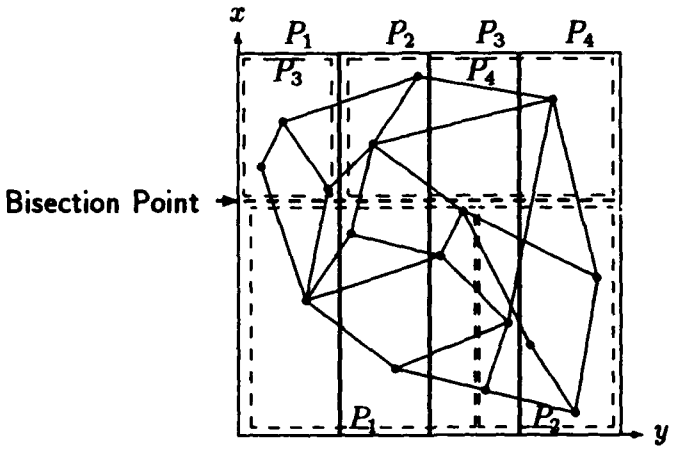

Figure 8: The graph of Figure 7 sorted by the y direction is also uniformly partitioned over the four processors. When a bisection is carried out in the x direction, as shown in Figure 7, this y graph must also be repartitioned. In this case all nodes with x coordinates less (greater) than the bisection point must be uniformly distributed across $P_1 - P_2(P_3 - P_4)$. Dashed boxes show the assignment of nodes after migration.

this case. Our objective in carrying out the migration is to ensure that the two halves of the processor chain, i.e. $P_1 - P_2$ & $P_3 - P_4$, contain the same subsets of nodes of the graph. Thus, since the nodes in Figure 7 to the *right* of the bisection point are moved to processors $P_3 - P_4$, we must similarly move the nodes *above* the bisection point in Figure 8 to $P_3 - P_4$. The nodes on the other side of the bisection point must conversely be sent to $P_1 - P_2$. The dashed boxes in Figure 8 show the ultimate destinations of the nodes of the y graph.

The interprocessor communication requirements of the y migration step are particularly severe. For example in Figure 8 we can see that P_2 and P_3 both need to send information to P_1. Similarly, P_3 and P_4 need to send information to P_2. It can also arise (although this is not illustrated in Figure 8) that one processor is required to send information out to several other processors. It is possible to satisfy this communication requirement using the complete exchange pattern, however it is possible to do much better, as the following discussion indicates.

Recall that our graph has been partitioned across a chain connected array of processors and the chain in turn embedded in a 2 or 3-d mesh or a hypercube using the BCE discussed earlier. Referring to Figure 8 we see that when several processors need to transmit to one processor, the transmitting processors form a subchain. For example, P_2, P_3, P_4 form a subchain that transmits to processor P_4. Instead of each processor transmitting individually to the destination, we can arrange to condense all information from a transmitting processor subchain into one processor, and then transmit from that one processor to the destination. Conversely, when one processor needs to receive from several processors, the receiving processors form a subchain and we can arrange to transmit to one of these processors and then disseminate this information to the recipients. The advantage in doing so is that the data movement between processors becomes a permutation and well-known techniques can be utilized for this [13, 22]. We shall discuss in Section 6.3 the details of these operations on specific interconnection structures. For the moment we shall assume that the time required for condensation and dissemination on a k processor system is $\mathcal{C}(k)$ and the time for permutation is $\mathcal{P}(k)$.

The process of y-migration is then as follows.

1. Condense all nodes of a subchain into one node,

2. transmit (permute) data between subchains, and

3. disseminate to nodes of receiving subchains.

In the worst case, each processor needs to send out or receive $O(n/p)$ points so that the time required for the y migration step is $O(n/p)\mathcal{P}(k)$. The total time for the x and y migrations is thus $O(n/p) + O(n/p)\mathcal{C}(k) + O(n/p)\mathcal{P}(k)$.

6.2.4 Summary

Table 2 summarizes the parallel algorithm and the time taken by each phase of the basic bisection step.

Table 2: Time taken by the basic bisection step.

1	Compute vectors	$O(n/p) + \mathcal{S}(k)$
2	Find optimal split point	$O(n/p) + \mathcal{M}(k)$
3	Migrate nodes	$O(n/p) + O(n/p)\mathcal{C}(k) + O(n/p)\mathcal{P}(k)$

6.3 Analysis of Run time

We now investigate the running time of the fast parallel algorithm on 2- and 3-d meshes and on hypercubes. This is done by substituting known expressions for the operations of Table 1 into the expressions of Table 2.

6.3.1 2-d meshes

Finding minimum and executing an arbitrary permutation on a 2-d mesh takes time $\mathcal{M}(k) = \mathcal{P}(k) = O(k^{1/2})$[13, 20]. The sum-prefix operation takes time $\mathcal{S}(k) = \log(k)\mathcal{P}(k) = \log(k)O(k^{1/2})$ as it requires $\log k$ different permutations.

To determine the time required for a condensation operation, we note that a BCE can only have aspect ratio 1 or 2. Thus a BCE with k nodes is mapped

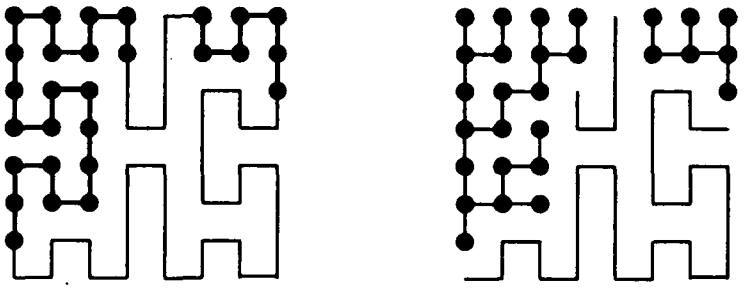

Figure 9: When a k node BCE is mapped onto a 2-d mesh, each subchain of the BCE can be spanned by a tree with degree constraint 4 and diameter less than the diameter of the mesh. Two subchains are indicated on the left of the diagram. The corresponding trees are shown on the right hand side.

---

onto a mesh of size $2^{\lceil \log_2 k \rceil} \times 2^{\lfloor \log_2 k \rfloor}$. This mesh has diameter $O(k^{1/2})$. It follows that the nodes of each subchain of a BCE can be connected by a spanning tree of diameter $O(k^{1/2})$, as illustrated in Figure 9. Furthermore, the degree of the nodes of this subtree is constrained to 4, since the mesh has degree 4. Thus the condensation operation can be carried out in parallel on all subchains in time $\mathcal{C}(k) = O(k^{1/2})$.

Recalling that $O(n/p)$ data points are transmitted at each step, the time for the basic bisection step (Table 2) is

$$\begin{aligned} & O(\frac{n}{p}) + \mathcal{S}(k) + \mathcal{M}(k) + O(\frac{n}{p})\mathcal{C}(k) + O(\frac{n}{p})\mathcal{P}(k) \\ &= \log k O(k^{1/2}) + O(k^{1/2}) + O(\frac{n}{p})O(k^{1/2}) + O(\frac{n}{p})O(k^{1/2}) \\ &= \log k O(k^{1/2}) + O(\frac{n}{p})O(k^{1/2}) \end{aligned}$$

This step is repeated for $k = p, p/2, p/4, \cdots$. The time for the entire fast parallel algorithm for an n node problem on a p processor system is thus

$$t_{\text{2-d mesh}} = O(\frac{n}{p^{1/2}} + p^{1/2} \log p). \tag{4}$$

6.3.2 3-d meshes

The time required to compute minimum, $\mathcal{M}(k)$, and to execute an arbitrary permutation, $\mathcal{P}(k)$, is $O(k^{1/3})$ [13, 20]. Sum prefix takes $\mathcal{S}(k) = \log k\mathcal{P}(k) = \log kO(k^{1/3})$. The condensation time $\mathcal{C}(k) = O(k^{1/3})$, using an argument similar to the one for 2-d meshes. The time for the basic bisection step is thus

$$\log kO(k^{1/3}) + O(\frac{n}{p})O(k^{1/3}).$$

The total time, obtained by summing the times for $k = p, p/2, p/4, \cdots$, is

$$t_{\text{3-d mesh}} = O(\frac{n}{p^{2/3}} + p^{1/3}\log p). \tag{5}$$

6.3.3 Hypercubes

On a hypercube, the time for computing minimum $\mathcal{M}(k) = O(\log k)$. The time for permuting data is $\mathcal{P}(k) = O(\log k)$ using Waksman's method[22], provided the required data routings are computed first. The $O(k \log k)$ overhead of this precomputation is prohibitive for permutations that are not known beforehand, as is the case for the node migration step (line 3 of Table 2).[8] We therefore use the simpler sorting approach to permuting data, which requires $\mathcal{P}(k) = O(\log^2 k)$ time and has no setup overhead.

For the computation of **leftvec** and **rightvec** (line 1 of table 2), we need to carry out a sum-prefix computation, which requires $\log k$ permutations, each with a fixed communication pattern. Waksman's method can be used in this case as the fixed data routings can be computed beforehand for a given size of hypercube. The time required for this operation is thus $\mathcal{S}(k) = \log^2 k$.

To investigate the condensation time $\mathcal{C}(k)$, we note that our domain has been mapped onto a BCE, which has been embedded in a 2- or 3-d mesh, which in turn has been embedded in a hypercube. Subchains of size k or less are thus wholly contained in subcubes of size $O(k)$ and can be spanned

[8] Nassimi and Sahni's algorithm[14], which also takes $O(\log k)$ time on hypercubes and requires no precomputation of routing, is restricted to a subset of all possible permutations. At this time, it is not known whether the data movements required in parallel parametric binary dissection fall into the category to which this technique can be applied.

by trees of diameter $O(\log k)$ with degree constraint $O(\log k)$, leading to $O(\log^2 k)$ condensation time. The times from table 2 become

$$\begin{aligned} & O(\frac{n}{p}) + \mathcal{S}(k) + \mathcal{M}(k) + O(\frac{n}{p})\mathcal{C}(k) + O(\frac{n}{p})\mathcal{P}(k) \\ = \; & O(\log^2 k) + O(\frac{n}{p})O(\log^2 k) + O(\frac{n}{p})O(\log^2 k) \\ = \; & O(\frac{n}{p})O(\log^2 k). \end{aligned}$$

The total time for hypercubes is thus

$$t_{\text{hypercube}} = O(\frac{n}{p}\log^3 p). \tag{6}$$

7 Applications to Unstructured Meshes

A portion of a 2-d unstructured mesh is shown in Figure 10. It can be seen that this mesh has a very large variation in node density. The objective, in generating the mesh, is to have a higher density of nodes in the regions where there is greater need for accuracy. It is this variation in density that makes such meshes difficult to partition. 3-dimensional unstructured meshes are an obvious extension but are impossible to illustrate on a 2-d page.

We have implemented the Fast Parametric Dissection algorithm of Section 4, using equation (2) of Section 3. This algorithm has been used to partition several very large 3-d unstructured grids taken from aerodynamic problems. When applying parametric dissection on such grids, it is often the case that the first cut is badly imbalanced as far as the number of nodes is concerned. This is because binary dissection considers the graph to be embedded in a rectangle or cuboid, with edges extending to the sides of the rectangle or cuboid (as shown in Figure 10). The mesh really occupies a roughly ellipsoidal region of 2 or 3-d space (which cannot be depicted in figure 10 as it is very large compared to the wing cross-section shown). When λ is non zero, the first cut is likely to slice off a small tip of the ellipsoid, so as to minimize the number of edges cut. Thus we have a tiny number of nodes in one region and most of the nodes in the other region. The objective (2) is correctly minimized and the partitioning obtained is superior to a plain partitioning, but only for depth 1. Beyond depth 1 or 2 this poor initial cut

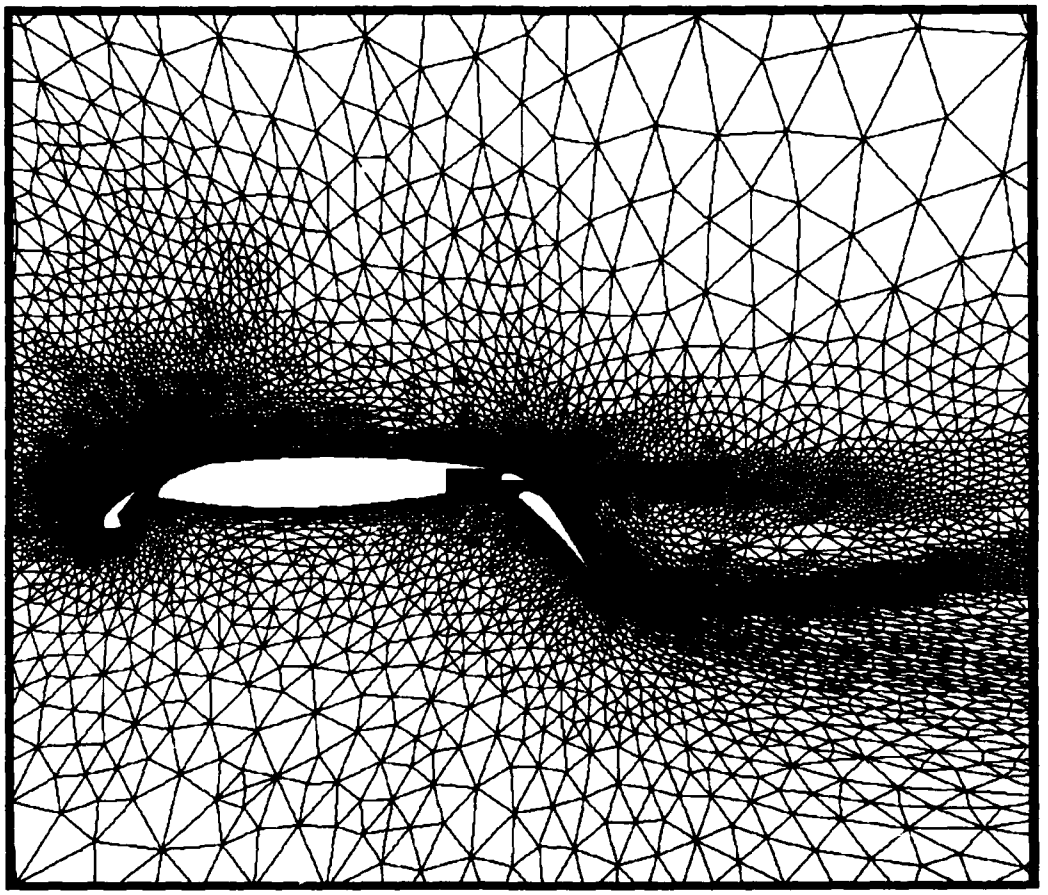

Figure 10: A 2-d unstructured mesh.

leads to bad partitions. This phenomenon is very similar to that described by Stone in connection with the partitioning of random graphs[18]. Our solution to this problem is to carry out the first 1, 2 or 3 cuts with $\lambda = 0$ and switch over to the desired value of λ only after these initial cuts have balanced the number of nodes in the initial 2, 4 or 8 subregions.

In order to evaluate the speedup that would be obtained if a parametric binary dissection were used, compared to plain binary dissection, we carried out an experiment with a 3-d mesh of size 106,064 nodes and 697,992 edges. This mesh is derived from a problem involving a wing and pod (engine enclosure) and half a fuselage. Measured run time on a 50 MHz MIPS R4000 processor for a depth 15 partition of this mesh is 83 seconds (excluding time to input the mesh).

The following evaluation procedure was repeated for depths $= 4 - 15$.

- Run the parametric dissection algorithm for $\lambda = 0.0, 0.2, \cdots, 1.0$.
- For each run obtain $maxnodes(\lambda)$ and $maxedges(\lambda)$, the maximum number of edges and nodes over all regions.
- The normalized run time for a dissection is

$$t_{\text{parametric}}(\lambda) = maxnodes(\lambda) + \lambda \times maxedges(\lambda).$$

  This assumes ideal communications on the target parallel processor.

- $maxnodes(0)$ and $maxedges(0)$ are the values that would have obtained if plain binary dissection had been used, since for $\lambda = 0$ parametric dissection reduces to plain dissection. Thus for this problem the time taken by a plain dissection is

$$t_{\text{plain}} = maxnodes(0) + \lambda \times maxedges(0).$$

- For a given value of λ the performance advantage of the parametric algorithm is

$$Improvement(\lambda) = \frac{t_{\text{plain}}}{t_{\text{parametric}}(\lambda)}.$$

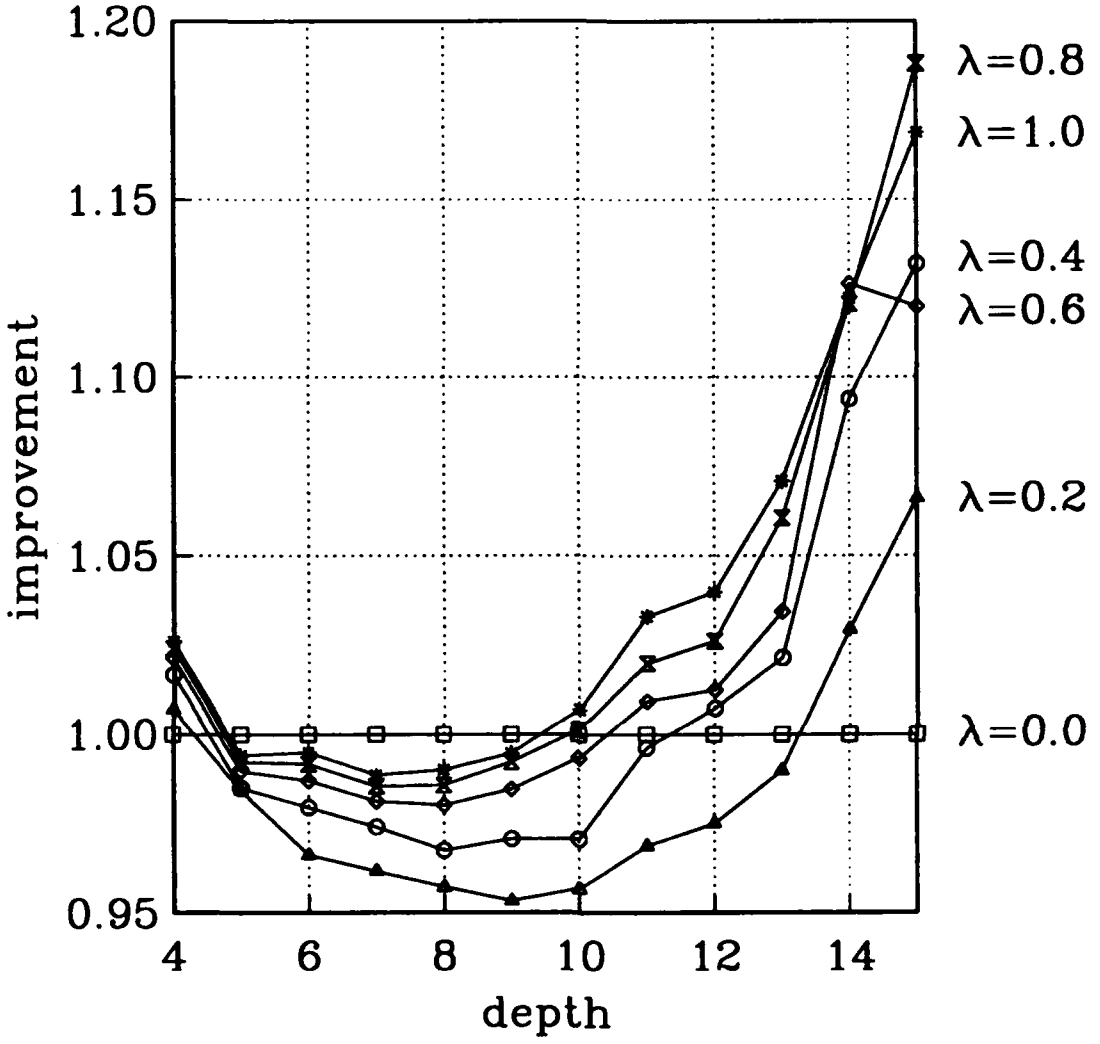

Figure 11: Improvement of parametric binary dissection over plain binary dissection, when applied to a 3-d aerodynamic mesh with ≈ 0.1 million nodes and ≈ 0.7 million edges, for depths 4-15 (corresponding to $16, 32, \cdots, 32768$ processors). For $\lambda = 0$ parametric dissection reduces to plain dissection and there is no speedup.

The results of the above experiment are summarized in the plots of Figure 11 which show the performance improvement of parametric dissection over plain dissection. Since parametric dissection reduces to plain dissection for $\lambda = 0$, the curve corresponding to this λ is constant at 1.00. There is no improvement for depth=1,2 or 3 because plain dissection is used for these depths, as discussed above, and these depths are not shown.

There is initially a small improvement at depth 4, after which performance is actually slightly poorer than plain dissection. Beyond depth=10, the advantage of using parametric dissection increases steeply with increasing depth. In this example the parametric algorithm yields partitions that are 20% better than plain dissection at depth 15.

8 Applications to Color Image Quantization

The algorithms described above have applications to partitioning problems unrelated to parallel processing. One example is *color image quantization*, in which samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error [11]. More formally, we are given a digital image whose pixels are chosen from a palette containing 2^m colors, and we wish to generate an acceptable reproduction using a palette of 2^k colors, where $k < m$. Typical values for m run from 15–24, while k is usually in the range from 8–12. Color quantization is commonly used to convert full-color images into *colormapped* or *pseudocolor* images in which each pixel is a k-bit index into a color lookup table, or *colormap*.

In full-color images, the m bits of color information are typically divided into three distinct color components, each using $\approx m/3$ bits. If we assume a red-green-blue (RGB) color model, then each component represents an axis in a three-dimensional color grid. The component values at each pixel can be thought of as indices into this grid. The problem then becomes one of partitioning the grid such that (1) the total number of regions is bounded by k, and (2) the maximum distance between any two points within a region is minimized. The latter constraint is a measure of the color error between the original image and the quantized result.

At the end of the partitioning process, the colors found in each region will map to the same representative value in the colormapped result. A variety of techniques have been proposed for partitioning the color space [10],

[11], [19], [23], [24]. Our approach most closely resembles Heckbert's *median cut* algorithm, but uses a modified version of Fast Dissection to speed up bounding box computations and reduce the maximum color error. We can also employ the parametric dissection idea to provide additional control over the placement of cuts.

The first step is to scan the original image and record which points in the color space are represented. We store this information in a 3-d Boolean matrix. The matrix is then scanned to produce a list of the colors which occur. The color list is replicated and sorted for each color component, as required by the Fast Dissection algorithm. Since our color components require only a few bits each, we can avoid the level of indirection required by the PARAMETRIC_CUT algorithm of Section 4. Instead, each color component can be stored directly as a bit field within a list item, reducing both memory and computation costs.

We next need a strategy for partitioning the lists. In the context of this problem, *load* (the number of colors in a region) is much less important than *shape* (the maximum distance between the points, or color error). Therefore our objective function is reformulated as $color\ error + \lambda \times (no.\ of\ colors)$. Increasing λ improves the ability to distinguish between colors in densely populated regions of the color space at the expense of poorer resolution in sparsely populated areas. Since the quality of a quantized image is often subjective, λ may be varied until the most pleasing result is achieved.

Other objective functions are certainly possible. For example, the colors in a partition could be weighted according to the number of times they occur in the original image (Heckbert's *popularity* criterion). This leads to a more accurate rendition of those colors which comprise large areas in the image, and less accurate rendition of others. Adjusting the value of λ determines how much the partitioning is influenced by the popularity counts. Simply setting λ to zero may be perfectly acceptable for many images, since this will tend to minimize the overall color error irrespective of other considerations.

For the sake of efficiency, we use a simple technique to estimate the color error in a region. Rather than searching for the two most extreme points and computing the distance between them, we use the length of the diagonal of the bounding box containing the points. With Fast Dissection, finding the bounding box is trivial—we simply obtain the respective maximum and minimum color components from each of the three sorted lists. At each partitioning step, these are accessible via the L and U list indices.

In our earlier descriptions of Parametric Binary Dissection and Fast Dissection, we have assumed a recursive partitioning process which descends until the maximum number of subregions is produced, or until a region cannot be subdivided further. For color quantization, we follow Heckbert's lead and modify this strategy to utilize *adaptive partitioning.* With adaptive partitioning, the directions of the cuts are not predetermined by the depth of the recursion, but are chosen dynamically based on properties of the data. A simple heuristic which works well for color quantization is to split the region along the longest edge, i.e., in the direction of largest color error. A more elaborate approach could use Fast Dissection to compute the split points in each direction, and evaluate the objective function for the resulting subregions. The cut would then be made in the most favorable direction.

One disadvantage of the recursive approach is that the partitioning process can "bottom out" prematurely—one or more branches of the recursion tree may encounter regions which cannot be further subdivided, even though other branches may offer ample opportunity for subdivision. The net result is that some of the available colormap entries go unused. Our solution to this problem uses an iterative variant of Fast Dissection. After each cut is made, the objective function is evaluated for the resulting subregions, and they are placed on a global subregion list, sorted by descending magnitude of the objective function. At each step of the iteration, the first subregion on the list is partitioned. This procedure guarantees that every available colormap entry will be used (assuming the original image contains at least 2^k colors), and it also drives the largest value of the objective function to a minimum.

When the partitioning phase is complete, the color of each region is set to the centroid of the bounding box, and all pixels whose original color lies in that region are mapped to the new value.

9 Conclusions

We have presented a new approach to the partitioning problem for non-uniform domains, analyzed its run time for serial and parallel machines and presented some measured performance figures. The parametric dissection algorithm is seen to provide better performance than the original binary dissection algorithm for large depths of partitioning.

A fast algorithm for parametric dissection was presented in Section 4. This algorithm has run time $O(n \log n)$ as opposed to the original $O(n \log^2 n)$ dissection algorithm. The time for dissection is thus completely masked by the time required to sort the input data.

We have presented two parallel algorithms for parametric dissection. The $O(n)$ algorithm is simple to implement and will likely be useful in situations where the mesh is being input serially to the processor, as in this case the dissection time is masked by the time to load. Our more elaborate algorithm has time $O((n/p^{1/2}) + p^{1/2} \log p)$, $O((n/p^{2/3}) + p^{1/3} \log p)$ and $O((n/p) \log^3 p)$ for 2-d meshes, 3-d meshes and hypercubes, respectively. This algorithm performs well for problems in which the number of nodes n is large compared to the number of processors, a case that is of considerable practical interest.

Future work in this area shall develop along the following lines.

1. Improvements in the parallel algorithm. Communication overhead, shows up prominently in the expressions for run time of our algorithm. Whether this can be reduced significantly is an open question.

2. Implementations of the parallel versions of the *dissection* algorithms for the iPSC-860, Touchstone Delta and Paragon.

3. Evaluation of the performance of PBD on a large set of unstructured meshes.

4. Use of these dissections for actual computation, especially for aerodynamic problems.

5. Applications of PBD to other areas, such as circuit and VLSI partitioning.

Acknowledgements

We are grateful to Dimitri Mavriplis for many useful discussions and for his enthusiastic support of this research. We thank Clyde Gumbert for providing us with a large 3-d mesh for experimentation. We are grateful to M. Y. Hussaini, K. E. Durrani, A. Hameed and S. Nazir Ahmad for their encouragement of this research.

References

[1] Krishan Belkhale and Prithviraj Banerjee. Recursive partitions on multiprocessors. In *The Fifth Distributed Memory Computing Conference*, pages 930–938, April 1990.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18:509–517, September 1975.

[3] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for pdes across multiprocessors. *Proceedings of the 1985 International Conference on Parallel Processing*, pages 166–170, August 1985.

[4] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems across multiprocessors. *IEEE Transactions on Computers*, C-36:570–580, May 1987.

[5] Manuel Blum, Robert W. Floyd, Vaughn Pratt, Ronald R. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[7] D. J. Mavriplis. Personal communication.

[8] D. J. Mavriplis. Three dimensional unstructured multigrid for the Euler equations. *AIAA Journal* 30(7):1753–1761, July 1992.

[9] Karen M. Dragon and John L. Gustafson. A low-cost hypercube load-balance algorithm. In *The Fifth Conference on Hypercubes, Concurrent Computers and Applications*, pages 583–589, March 1989.

[10] Michael Gervautz and Werner Purgathofer. A simple method for color quantization: octree quantization. *Graphics Gems*, 287–293, A. Glassner, ed., Academic Press, 1990.

[11] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics*, 16:297–307, July 1982.

[12] G. Miller, S-H. Teng, W. Thurston and S. Vavasis. Automatic mesh partitioning. Technical Report CTC92TR112, Cornell Theory Center, 1992.

[13] David Nassimi and Sartaj Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *CACM*, 29:642–677, July 1982.

[14] David Nassimi and Sartaj Sahni. A self-routing Benes network and parallel permutation algorithms. *IEEE Trans. Computers*, C-30(5):332–340, May 1981.

[15] A. Pothen, H. D. Simon and K.P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications"*, 11:430–452, 1990.

[16] Youcef Saad and Martin H. Schultz. Topological properties of hypercubes. *IEEE Trans. Computers*, C-37(7):867–872, 1988.

[17] J. Saltz, S. Petiton, H. Berryman and A. Rifkin. Performance effects of irregular communications on massively parallel multiprocessors. *JPDC*, 13(2):202–212, 1991.

[18] Harold S. Stone. *High Performance Computer Architecture*. Addison-Wesley, Reading, Massachusetts, 1990, pp. 311-313.

[19] Spencer W. Thomas. Efficient inverse color map computation. *Graphics Gems II*, 116–125, J. Arvo, ed., Academic Press, San Diego, CA, 1991.

[20] C. D. Thompson and H. T. Kung. Sorting on a mesh connected parallel computer. *CACM*, 20:263–271, April 1977.

[21] S. Vavasis. Automatic domain partitioning in three dimensions. *SIAM Journal on Scientific and Statistical Computing*, 12:950–970, 1991.

[22] Abraham Waksman. A permutation network. *JACM*, 15:159–163, January 1968.

[23] S. Wan, S. Wong, and P. Prusinkiewicz, An algorithm for multidimensional data clustering, *ACM Transactions on Mathematical Software*, 14(2):153-162, 1968.

[24] Xiaolin Wu. Efficient statistical computations for optimal color quantization. *Graphics Gems II*, 126–133, J. Arvo, ed., Academic Press, San Diego, CA, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1993	Contractor Report

4. TITLE AND SUBTITLE
PARAMETRIC BINARY DISSECTION

5. FUNDING NUMBERS
C NAS1-19480
C NAS1-18605
WU 505-90-52-01

6. AUTHOR(S)
Shahid H. Bokhari, Thomas W. Crockett, and
David M. Nicol

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 93-39

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-191496
ICASE Report No. 93-39

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report
Submitted to IEEE Transactions on Computers

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 63, 66

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Binary dissection is widely used to partition non-uniform domains over parallel computers. This algorithm does not consider the perimeter, surface area, or aspect ratio of the regions being generated and can yield decompositions that have poor communication to computation ratio.

Parametric Binary Dissection (PBD) is a new algorithm in which each cut is chosen to minimize $load + \lambda \times (shape)$. In a 2 (or 3) dimensional problem, *load* is the amount of computation to be performed in a subregion and shape could refer to the perimeter (respectively surface) of that subregion. *Shape* is a measure of communication overhead and the parameter λ permits us to trade off load imbalance against communication overhead. When λ is zero, the algorithm reduces to plain binary dissection.

This algorithm can be used to partition graphs embedded in 2 or 3-d. Here load is the number of nodes in a subregion, shape the number of edges that leave that subregion, and λ the ratio of time to communicate over an edge to the time to compute at a node. We present an algorithm that finds the depth d parametric dissection of an embedded graph with n vertices and e edges in $O(\max[n \log n, de])$ time, which is an improvement over the $O(dn \log n)$ time of plain binary dissection. We also present parallel versions of this algorithm; the best of these requires $O((n/p) \log^3 p)$ time on a p processor hypercube, assuming graphs of bounded degree.

We describe how PBD is applied to 3-d unstructured meshes and yields partitions that are better than those obtained by plain dissection. We also discuss its application to the *color image quantization* problem, in which samples in a high-resolution color space are mapped onto a lower resolution space in a way that minimizes the color error.

14. SUBJECT TERMS	15. NUMBER OF PAGES
binary dissection; color image quantization; graph partitioning; load balancing; orthogonal recursive bisection; parametric binary dissection; partitioning; parallel processing	37
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

☆U.S. GOVERNMENT PRINTING OFFICE: 1993 - 728-064/86034